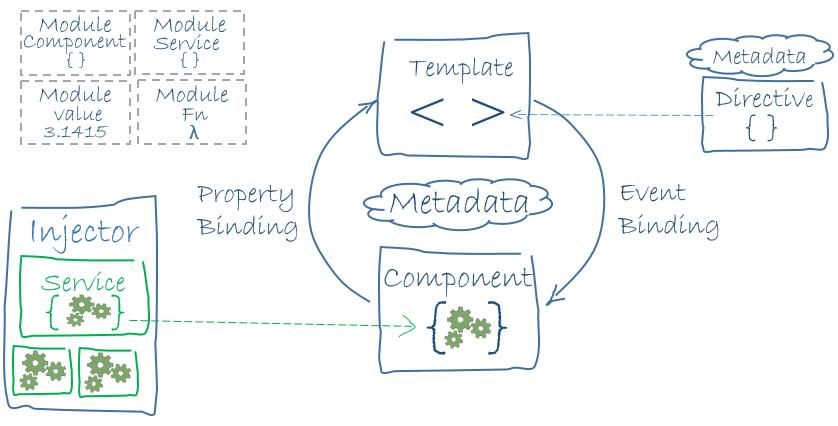
# Architecture

ith Angular, we write applications by composing HTML templates with Angularized-markup, writing component classes to manage those templates, adding application logic in services, and handing the top root component to Angular's bootstrapper.



## Modules

Angular apps are modular. In general we assemble our application from many modules.

app.component.ts

export class AppComponent{…}

The export statement tells TypeScript that this is a module whose AppComponent class is public and accessible to other modules of the application.

### Import from local modules

When we need a reference to the AppComponent, we **import** it like this:

app/main.ts (import)

import { AppComponent } from './app.component';

The import statement tells the system it can get an AppComponent from a module named app.component located in a neighboring file. The module name (AKA module id) is often the same as the filename without its extension.

### Import from Libraries

Some modules are libraries of other modules. Angular itself ships as a collection of library modules within several npm packages. Their names begin with the @angular prefix.

app/app.component.ts (import)

import { Component } from '@angular/core';

## Component

A component controls a patch of screen real estate that we could call a view. The shell at the application root with navigation links, that list of heroes, the hero editor ... they're all views controlled by components.

We define a component's application logic — what it does to support the view — inside a class. The class interacts with the view through an API of properties and methods.

A Component has a lifecycle managed by Angular itself. Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them.

|  |  |
| --- | --- |
| **Hook** | **Purpose** |
| ngOnInit | Initialize the directive/component after Angular initializes the data-bound input properties. |
| ngOnChanges | Respond after Angular sets a data-bound input property. The method receives achanges object of current and previous values. |
| ngDoCheck | Detect and act upon changes that Angular can or won't detect on its own. Called every change detection run. |
| ngOnDestroy | Cleanup just before Angular destroys the directive/component. Unsubscribe observables and detach event handlers to avoid memory leaks. |

**Components only:**

|  |  |
| --- | --- |
| **Hook** | **Purpose** |
| ngAfterContentInit | After Angular projects external content into its view. |
| ngAfterContentChecked | After Angular checks the bindings of the external content that it projected into its view. |
| ngAfterViewInit | After Angular creates the component's view(s). |
| ngAfterViewChecked | After Angular checks the bindings of the component's view(s). |

### Lifecycle Sequence

Angular calls the lifecycle hook methods in the following sequence at specific moments:

|  |  |
| --- | --- |
| **Hook** | **Timing** |
| ngOnChanges | before ngOnInit and when a data-bound input property value changes. |
| ngOnInit | after the first ngOnChanges. |
| ngDoCheck | during every Angular change detection cycle. |
| ngAfterContentInit | after projecting content into the component. |
| ngAfterContentChecked | after every check of projected component content. |
| ngAfterViewInit | after initializing the component's views and child views. |
| ngAfterViewChecked | after every check of the component's views and child views. |
| ngOnDestroy | just before Angular destroys the directive/component. |

The router component, for instance, has it’s own router lifecycle hooks that allow us to tap into specific moments in route navigation.

## Metadata

Metadata tells Angular how to process a class. In TypeScript, we attach metadata by using a decorator. Here's some metadata for HeroListComponent:

@Component({

selector: 'hero-list',

templateUrl: 'app/hero-list.component.html',

directives: [HeroDetailComponent],

providers: [HeroService]

})

export class HeroListComponent implements OnInit {

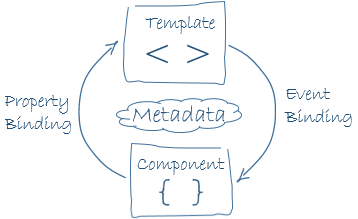
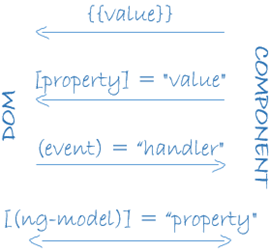
/\* . . . \*/

}

**A decorator is a function.** Decorators often have a configuration parameter. The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

* directives: array of the components or directives that this template requires. We saw in the last line of our template that we expect Angular to insert a HeroDetailComponent in the space indicated by <hero-detail> tags. Angular will do so only if we mention the HeroDetailComponent in this directives array.
* provides: array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that our component's constructor requires a HeroService so it can get the list of heroes to display.

## Data binding



<li>{{hero.name}}</li>

<hero-detail [hero]="selectedHero"></hero-detail>

<li (click)="selectHero(hero)"></li>

* Interpolation: The {{hero.name}} interpolation displays the component's hero.name property value within the <li> tags.
* Property binding: The [hero] property binding passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
* Event binding: The (click) event binding calls the component's selectHero method when the user clicks a hero's name.
* Two-way data binding: using the ngModel directive.

## Services

Service is a broad category encompassing any value, function, or feature that our application needs.

We prefer our component classes lean. Our components don't fetch data from the server, they don't validate user input, and they don't log directly to the console. They delegate such tasks to services.

Angular doesn't enforce these principles. It won't complain if we write a "kitchen sink" component with 3000 lines.

Angular does help us follow these principles by making it easy to factor our application logic into services and make those services available to components through dependency injection.

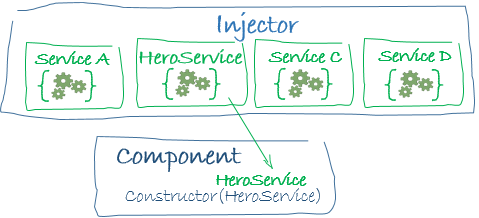
## Dependency Injection

Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

Angular can tell which services a component needs by looking at the types of its constructor parameters. For example, the constructor of our HeroListComponent needs a HeroService:

app/hero-list.component.ts (constructor)

constructor(private service: HeroService) { }



# Displaying Data

## Showing component properties with interpolation

@Component({  
 **selector**: **'my-app'**,  
 **template**: **`  
 <h1>{{title}}</h1>  
 <h2>My favorite hero is: {{myHero}}</h2>  
 `**})  
*// More compact compare with constructor***export class** AppComponent {  
 **title** = **'Tour of Heroes'**;  
 **myHero** = **'Windstorm'**;  
}  
*// or initialize variable in constructor***export class** AppCtorComponent {  
 **title**: **string**;  
 **myHero**: **string**;  
 **constructor**() {  
 **this**.**title** = **'Tour of Heroes'**;  
 **this**.**myHero** = **'Windstorm'**;  
 }  
}

If a property is showing in the view, Changing the value of a property will trigger a corresponding update in the view, the showing value will be updated.

@Component({  
 **selector**: **'my-app'**,  
 **template**: **'<h1>My First Angular 2 App</h1><h2>It is the {{*name*}}</h2>'**})  
**export class** AppComponent {  
 **name**: **string** =**"hello"**;  
 **constructor**(){  
 *setTimeout*(()=>{  
 **this**.**name** = **"hello world!"**; *// after 5 seconds showing new value: "hello world!".* }, 5000);  
 }  
}

## Showing an array prpoerty with \*ngFor

@Component({  
 **selector**:**'my-app'**,  
 **template**: **`   
 <h1>{{title}}</h1>  
 <h2>My favorite hero is: {{myHero}}</h2>  
 <p>Heroes:</p>  
 <ul>  
 <li \*ngFor="let hero of heroes">  
 {{ hero }}  
 </li>  
 </ul>`**})  
**export class** AppComponent {  
 **title** = **'Tour of Heroes'**;  
 **heroes** = [**'Windstorm'**, **'Bombasto'**, **'Magneta'**, **'Tornado'**];  
 **myHero** = **this**.**heroes**[0];  
}

## Conditional display with \*ngIf

**<p \*ngIf="heroes.length > 3">There are many heroes!</p>**

Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That hardly matters here. But it would matter a great deal, from a performance perspective, if we were conditionally including or excluding a big chunk of HTML with many data bindings.

# User Input

User input triggers DOM events. We listen to those events with event bindings that funnel updated values back into our components and models.

When the user clicks a link, pushes a button, or enters text we want to know about it. These user actions all raise DOM events. he syntax is simple. We surround the DOM event name in parentheses and assign a quoted template statement to it.

**<button (click)="onClickMe()">Click me!</button>**

The (click) to the left of the equal sign identifies the button's click event as the **target of the binding**. The text within quotes on the right is the **template statement** in which we respond to the click event by calling the component's onClickMe method. A [template statement](#_Template_Statements) is a subset of JavaScript with restrictions and a few added tricks.

When writing a binding we must be aware of a template statement's **execution context**. That is usually the Angular component that controls the template .

@Component({  
 **selector**: **'click-me'**,  
 **template**: **`<button (click)="onClickMe()">Click me!</button>{{clickMessage}}`**})  
**export class** ClickMeComponent {  
 **clickMessage** = **''**;  
 onClickMe() {  
 **this**.**clickMessage** = **'You are my hero!'**;  
 }  
}

When the user clicks the button, Angular calls the component's onClickMe method.

## Get user input from the $event object

**template**: **`  
 <input (keyup)="onKey($event)">  
 <p>{{***values***}}</p>  
`**

Angular makes an event object available in the $event variable, which we pass to the component's onKey() method.

**export class** KeyUpComponent\_v1 {  
 **values** = **''**;  
 *// without strong typing* onKey(event:**any**) {  
 **this**.**values** += event.**target**.**value** + **' | '**;  
 }  
}

he shape of the $event object is determined by whatever raises the event. The keyup event comes from the DOM, so $event must be a standard DOM event object. The $event.target gives us in this example an HTMLInputElement, which has a **value** property that contains our user input data.

We generally prefer the strong typing that TypeScript affords. We can rewrite the method, casting to HTML DOM objects like this.

**export class** KeyUpComponent\_v1 {  
 **values** = **''**;  
  
 *// with strong typing* onKey(event: KeyboardEvent) {  
 **this**.**values** += (<HTMLInputElement>event.**target**).**value** + **' | '**;  
 }  
}

Strong typing reveals a serious problem with passing a DOM event into the method: too much awareness of template details, too little separation of concerns.

## Get user input from a template reference variable

Here’s an example of using a template reference variable to implement a clever keystroke loopback in an ultra-simple template:

@Component({  
 **selector**: **'loop-back'**,  
 **template**: **`  
 <input #box (keyup)="0">  
 <p>{{*box*.value}}</p>  
 `**})  
**export class** LoopbackComponent { }

**But this won't work at all unless we bind to an event.**

Angular only updates the bindings (and therefore the screen) if we do something in response to asynchronous events such as keystrokes.

That's why we bind the keyup event to a statement that does ... well, nothing. We're binding to the number 0, the shortest statement we can think of. That is all it takes to keep Angular happy. We said it would be clever!

@Component({  
 **selector**: **'key-up2'**,  
 **template**: **`  
 <input #box (keyup)="onKey(*box*.value)">  
 <p>{{values}}</p>  
 `**})  
**export class** KeyUpComponent\_v2 {  
 **values** = **''**;  
 onKey(value: **string**) {  
 **this**.**values** += value + **' | '**;  
 }  
}

An especially nice aspect of this approach is that our component code gets clean data values from the view. It no longer requires knowledge of the $event and its structure.

## Key event Filtering (with key.event)

Perhaps we don't care about every keystroke. Maybe we're only interested in the input box value when the user presses Enter, and we'd like to ignore all other keys.

When we bind to the (keyup) event, our event handling statement hears every keystroke. We could filter the keys first, examining every $event.keyCode, and update the values property only if the key is Enter.

Angular can filter the key events for us. Angular has a special syntax for keyboard events. We can listen for just the Enter key by binding to Angular's keyup.enter pseudo-event.

@Component({  
 **selector**: **'key-up3'**,  
 **template**: **`  
 <input #box (keyup.enter)="values=*box*.value">  
 <p>{{values}}</p>  
 `**})  
**export class** KeyUpComponent\_v3 {  
 **values** = **''**;  
}

## On blur

@Component({  
 **selector**: **'key-up4'**,  
 **template**: **`  
 <input #box  
 (keyup.enter)="values=*box*.value"  
 (blur)="values=*box*.value">  
  
 <p>{{values}}</p>  
 `**})  
**export class** KeyUpComponent\_v4 {  
 **values** = **''**;  
}

# Template Syntax

## Template Statements

A template statement responds to an event raised by a binding target such as an **element**, **component**, or **directive**.

(<event>)="<statement>"

A template statement has a side effect. It's how we update application state from user input. There would be no point to responding to an event otherwise.

Like template expressions, template statements use a language that looks like JavaScript. The template statement parser is different than the template expression parser and specifically supports both basic assignment (=) and chaining expressions (with ; or ,).

However, certain JavaScript syntax is not allowed:

* New
* increment and decrement operators, ++ and –
* operator assignment, such as += and -=
* the bitwise operators | and &
* the template expression operators

As with expressions, statements can refer only to what's in the statement context — typically the **component instance** to which we're binding the event.

Template statements cannot refer to anything in the global namespace. They can’t refer to window or document.

The statement context may include an object other than the component. E.g. a [template reference variable](#_Template_reference_variables).

## Template reference variables

A template reference variable is a reference to a DOM element or directive within a template.

We can reference a template reference variable on the same element, on a sibling element, or on any child elements.

Here are two other examples of creating and consuming a Template reference variable:

*<!-- phone refers to the input element; pass its `value` to an event handler -->*<**input #phone placeholder="phone number"**>  
<**button (click)="callPhone(*phone*.value)"**>Call</**button**>  
  
*<!-- fax refers to the input element; pass its `value` to an event handler -->*<**input ref-fax placeholder="fax number"**>  
<**button (click)="callFax(fax.value)"**>Fax</**button**>

The hash (#) prefix to "phone" means that we're defining a phone variable. Folks who don't like using the # character can use its canonical alternative, the ref- prefix. For example, we can declare the our phone variable using either #phone or ref-phone.

Another example: ngForm and template reference variables. The HTML for a form can be quite involved.

<**form (ngSubmit)="onSubmit(*theForm*)" #theForm="ngForm"**>  
 <**div class="form-group"**>  
 <**label for="name"**>Name</**label**>  
 <**input class="form-control" required ngControl="firstName"  
 [(ngModel)]="currentHero.firstName"**>  
 </**div**>  
 <**button type="submit" [disabled]="!*theForm*.form.valid"**>Submit</**button**>  
</**form**>

What is the value of theForm?

It would be the HTMLFormElement if Angular hadn't taken it over. It's actually ngForm, a reference to the Angular built-in NgForm directive that wraps the native HTMLFormElement and endows it with additional superpowers such as the ability to track the validity of user input.

This explains how we can disable the submit button by checking theForm.form.valid and pass an object with rich information to the parent component's onSubmit method.

# Forms

## binding

If you use input-tag inside of a form, the name attribut must be defined. This is a requirement when using [(ngModel)] in combination with a form, so that we can easily refer to it in the aggregate form value and validity state.

<**input id="name" type="text" class="form-control" required [(ngModel)]="model.name"**>

The following error is shown:

ORIGINAL EXCEPTION: If ngModel is used within a form tag, either the name attribute must be set or the form control must be defined as 'standalone' in ngModelOptions.

<**input id="name" type="text" class="form-control" required [(ngModel)]="model.name" name="name"**>

The punctuation in the binding syntax**, [()]**, is a good clue to what's going on.

In a Property Binding, a value flows from the model to a target property on screen. We identify that target property by surrounding its name in brackets, []. This is a **one-way data binding from the model to the view**.

In an Event Binding, we flow the value from the target property on screen to the model. We identify that target property by surrounding its name in parentheses, (). This is a one-way data binding in **the opposite direction from the view to the model**.

No wonder Angular chose to combine the punctuation as [()] to signify a two-way data binding and a flow of data in both directions.

In fact, we can break the NgModel binding into its two separate modes as we do in this re-write of the "Name" <input> binding:

**<input type="text" class="form-control" required [ngModel]="model.name" (ngModelChange)="model.name = $event" >**

The **ngModelChange** is not an <input> element event. It is actually an event property of the NgModel directive. When Angular sees a binding target in the form [(x)], it expects the x directive to have an **x** input property and an **xChange** output property.

The other oddity is the template expression, model.name = $event. We're used to seeing an $event object coming from a DOM event. The ngModelChange property doesn't produce a DOM event; it's an Angular EventEmitter property that returns the input box value when it fires — which is precisely what we should assign to the model's name property.

We almost always prefer [(ngModel)]. We might split the binding if we had to do something special in the event handling such as debounce or throttle the key strokes.

## Validation

A form isn't just about data binding. We'd also like to know the state of the controls on our form. Using ngModel in a form gives us more than just two way data binding. It also tells us if the user touched the control, if the value changed, or if the value became invalid.

**ngModel** doesn't just track state; it updates the control with special Angular CSS classes from the set we listed above. We can leverage those class names to change the appearance of the control and make messages appear or disappear.

Internally Angular creates FormControls and registers them with an NgForm directive that Angular attached to the <form> tag. Each FormControl is registered under the name we assigned to the name attribute.

The NgModel directive doesn't just track state. It updates the control with three classes that reflect the state.

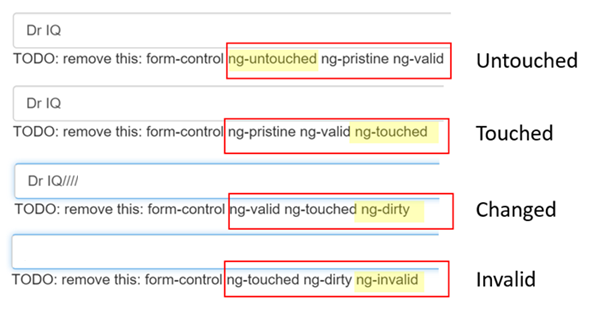
|  |  |  |
| --- | --- | --- |
| **State** | **Class if true** | **Class if false** |
| Control has been visited | ng-touched | ng-untouched |
| Control's value has changed | ng-dirty | ng-pristine |
| Control's value is valid | ng-valid | ng-invalid |

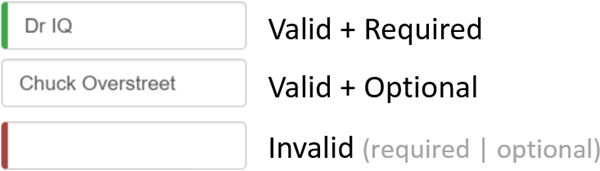
<input type="text" class="form-control" required

[(ngModel)]="model.name"

name="name" #spy >

<br>TODO: remove this: {{spy.className}}





We achieve this effect by adding two styles to a new forms.cssfile that we add to our project as a sibling to index.html.

forms.css

.ng-valid[required] {

border-left: 5px solid #42A948; /\* green \*/

}

.ng-invalid {

border-left: 5px solid #a94442; /\* red \*/

}

## Show and Hide Validation Error messages

We need a template reference variable to access the input box's Angular control from within the template. Here we created a variable called name and gave it the value "ngModel".

<**label for="name"**>Name</**label**>  
<**input type="text" class="form-control" required  
 [(ngModel)]="model.name"  
 name="name" #name="ngModel"** >  
<**div [hidden]="*name*.valid || *name*.pristine" class="alert alert-danger"**>  
 Name is required  
</**div**>

We need a template reference variable to access the input box's Angular control from within the template. Here we created a variable called **name** and gave it the value "**ngModel**".

<**input id="name" type="text" class="form-control" required [(ngModel)]="model.name" name="name" #name="ngMod"**>

There is no directive with "exportAs" set to "ngMod" (" <input id="name" type="text" class="form-control" required [(ngModel)]="model.name" name="name" [ERROR ->]#name="ngMod">

Why "ngModel"? A directive's exportAs property tells Angular how to link the reference variable to the directive. We set name to ngModel because the ngModel directive's exportAs property happens to be "ngModel".

<div [hidden]="name.valid || name.pristine" class="alert alert-danger">

In this example, we hide the message when the control is valid or pristine; pristine means the user hasn't changed the value since it was displayed in this form.

<**form (ngSubmit)="onSubmit()" #heroForm="ngForm"**>

**ngSubmit** is an Angular directive and bind it to HeroFormComponent.onSubmit method with an event binding.

We defined a template reference variable, **#heroForm**, and initialized it with the value, "ngForm". The variable **heroForm** is now a reference to the **NgForm** **directive** that governs the form as a whole.

What **NgForm** directive? We didn't add an NgForm directive! Angular did. Angular creates and attaches an NgForm directive to the <form> tag automatically.

The NgForm directive supplements the form element with additional features.

* It holds the controls we created for the elements with ngModel directive and name attribute and monitors their properties including their validity.
* It also has its own valid property which is true only if every contained control is valid.

<**button type="submit" class="btn btn-default" [disabled]="!*heroForm*.form.valid"**>Submit</**button**>

**Toggle two form regions:**

<div [hidden]="submitted">

<h1>Hero Form</h1>

<form \*ngIf="active" (ngSubmit)="onSubmit()" #heroForm="ngForm">

<!-- ... all of the form ... -->

</form>

</div>

submitted = false;

onSubmit() { this.submitted = true; }

When we click the Submit button, the submitted flag becomes true and the form disappears as planned.

# Routing

HTML5 introduced the history.pushState() and history.replaceState() methods, which allow you to add and modify history entries, respectively.

Thanks to pushState, we can make our in-app URL paths look the way we want them to look, e.g. localhost:3000/crisis-center. Our in-app URLs can be indistinguishable from server URLs. Modern HTML 5 browsers were the first to support pushState which is why many people refer to these URLs as "HTML 5 style" URLs.

We must add a <base href> element tag to the index.html to make pushState routing work. The browser also needs the base href value to prefix relative URLs when downloading and linking to css files, scripts, and images.

Add the base element just after the <head> tag. If the app folder is the application root, as it is for our application, set the href value in index.html exactly as shown here.

<base href="/">

## Configure and add the router

**import** {RouterConfig, *provideRouter*} **from "@angular/router"**;  
**import** {HeroesComponent} **from "./heroes.component"**;  
**const** routes:RouterConfig = [{  
 **path**: **'heores'**,  
 **component**: HeroesComponent  
 }];  
**export const** appRouterProviders = [*provideRouter*((routes))];

The RouterConfig is an array of route definitions.

This route definition has two parts:

* **path**: the router matches this route's path to the URL in the browser address bar (/heroes).
* **component**: the component that the router should create when navigating to this route (HeroesComponent).

## Make the router available

We have to import our appRouterProviders which contains our configured router and make it available to the application by adding it to the bootstrap array.

**import** { *bootstrap* } **from '@angular/platform-browser-dynamic'**;  
**import** { AppComponent } **from './app.component'**;  
**import** {appRouterProviders} **from "./app.routes"**;  
  
*bootstrap*(AppComponent, [appRouterProviders]);

## Router Outlets and Router Links

If we paste the path, /heroes, into the browser address bar, the router should match it to the 'Heroes' route and display the HeroesComponent. We have to tell it where by adding <router-outlet> marker tags to the bottom of the template.

RouterOutlet is one of the ROUTER\_DIRECTIVES. The router displays each component immediately below the <router-outlet> as we navigate through the application.

Normally we add additional an anchor tag to the template which triggers naviation to.

template: `

<h1>{{title}}</h1>

<a [routerLink]="['/heroes']">Heroes</a>

<router-outlet></router-outlet>

`

The [routerLink] binding in the anchor tag. We bind the RouterLink directive (another of the ROUTER\_DIRECTIVES) to an array that tells the router where to navigate when the user clicks the link.

If we want the app to show the dashboard when it starts and we want to see a nice URL in the browser address bar that says /dashboard.

{

path: '',

redirectTo: '/dashboard',

pathMatch: 'full'

},

We can also configure a route with a parameter:

{

path: 'detail/:id',

component: HeroDetailComponent

},

The colon (:) in the path indicates that :id is a placeholder to be filled with a specific hero id when navigating to the HeroDetailComponent.

# Two-way Binding with NgModel

When developing data entry forms, we often want to both display a data property and update that property when the user makes changes.

The [(ngModel)] two-way data binding syntax makes that easy. Here's an example:

<input [(ngModel)]="currentHero.firstName">

[()] = BANANA IN A BOX

To remember that the parentheses go inside the brackets, visualize a banana in a box.

Alternatively, we can use the canonical prefix form:

<input bindon-ngModel="currentHero.firstName">

We could have achieved the same result with separate bindings to the <input> element's value property and input event. That’s however cumbersome.

<input [value]="currentHero.firstName" (input)="currentHero.firstName=$event.target.value" >

That ngModel directive hides these onerous details behind its own ngModel input and ngModelChange output properties.

<input [ngModel]="currentHero.firstName" (ngModelChange)="currentHero.firstName=$event">

*The ngModel input property sets the element's value property and the ngModelChange output property listens for changes to the element's value. The details are specific to each kind of element and therefore the NgModel directive only works for elements, such as the input text box, that are supported by a ControlValueAccessor. We can't apply [(ngModel)] to our custom components until we write a suitable value accessor.*

We shouldn't have to mention the data property twice. Angular should be able to capture the component’s data property and set it with a single declaration — which it can with the [( )] syntax:

<input [(ngModel)]="currentHero.firstName">

### Syntactic suger[(x)]

[(x)] is just syntactic sugar for a property binding and an event binding:

[x]="someParentProperty" (xChange)="someParentProperty=$event"

To achive the following code:

<my-comp [(myText)]="testString"></my-comp>

In the component for my-comp must have an myText property and an myTextChange property(which is a EventEmitter).

**export class** MyComp {  
 @Input() **myText**: **string**;  
 @Output() **myTextChange**: EventEmitter<**string**> = **new** EventEmitter();

*//Notify parent of changes, whenever the value of myText changes, emit an event.* onChange(newMyText:**string**) {  
 **this**.**myTextChange**.emit(newMyText);  
 }  
}

Another example:

In the super component:

<my-hero-detail [hero]="selectedHero" [(age)]="heroAge" />

export class HeroesComponent {private age:number=20}

In the sub component:

**<input [(ngModel)]="heroage" placeholder="0"/>**

Export class HeroDetailComponent{

@Input()  
 **heroAge**:**number**;

@Output()  
 **heroageChange**:EventEmitter<**number**> = **new** EventEmitter<**number**>();  
  
 onChange() {  
 **this**.**ageChange**.emit(**this**.age);  
 }

}

### Aliasing input/output properties

Sometimes we want the public name of an input/output property to be different from the internal name.

This is frequently the case with [attribute directives](#_Attribute_directives). Directive consumers expect to bind to the name of the directive. For example, when we apply a directive with a myClick selector to a <div> tag, we expect to bind to an event property that is also called myClick.

<div (myClick)="clickMessage=$event">click with myClick</div>

However, the directive name is often a poor choice for the name of a property within the directive class. The directive name rarely describes what the property does. The myClick directive name is not a good name for a property that emits click messages.

Fortunately, we can have a public name for the property that meets conventional expectations, while using a different name internally.

@Output('myClick') clicks = new EventEmitter<string>(); // @Output(alias) propertyName = ...

Or alternative:

@Directive({

outputs: ['clicks:myClick'] // propertyName:alias

})

# Directives

There are three kinds of directives in Angular:

* Components
* Structural directives
* Attribute directives

A Component is really *a directive with a template*. It's the most common of the three directives and we tend to write lots of them as we build applications.

Structural directives can change the DOM layout by adding and removing DOM elements. **NgFor** and **NgIf** are two familiar examples.

An Attribute directive can change the appearance or behavior of an element. The built-in **NgStyle** directive, for example, can change several element styles at the same time.

We don't need any directive to simply set the background color. We can set it with the special Style Binding like this:

<p [*style.background*]="'lime'">I am green with envy!</p>

## Attribute directives

An attribute directive minimally requires building a controller class annotated with @Directive, which specifies the selector identifying the attribute associated with the directive. The controller class implements the desired directive behavior.

An example highlight.directive.ts:

**import** {Directive, ElementRef} **from "@angular/core"**;  
@Directive({  
 **selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
 **constructor**(el: ElementRef){  
 el.**nativeElement**.**style**.**backgroundColor** = **'yellow'**;  
 }  
}

We need the ElementRef to inject into the directive's constructor so we can access the DOM element.

@Directive requires a CSS selector to identify the HTML in the template that is associated with our directive. The CSS selector for an attribute is the attribute name in square brackets. Our directive's selector is *[myHighlight]*.

We export `HighlightDirective` to make it accessible to other components.

**import** {Component} **from "@angular/core"**;  
**import** {HighlightDirective} **from "./highlight.directive"**;  
@Component({  
 **selector**: **'my-app'**,  
 **directives**: [HighlightDirective],  
 **template**: **'<h1>My First Attribute Directive</h1><p myHighlight>Highlight me!</p>'**})

Angular creates a new instance of the directive's controller class for each matching element, injecting an Angular ElementRef into the constructor. ElementRef is a service that grants us direct access to the DOM element through its nativeElement property. That's all we need to set the element's background color using the browser DOM API.

### Respond to user action

Our directive should be able in response to a user action. We apply the @HostListener decorator to methods which are called when an event is raised.

@HostListener('mouseenter') onMouseEnter() {/\* . . . \*/}

@HostListener('mouseleave') onMouseLeave() {/\* . . . \*/}

**import** {Directive, ElementRef, HostListener} **from "@angular/core"**;  
@Directive({  
 **selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
 **constructor**(**private** el: ElementRef){}  
 @HostListener(**'onmouseenter'**)  
 **private** onMouseEnter(){  
 **this**.hightlight(**'yellow'**);  
 }  
 @HostListener(**'onmouseleave'**)  
 **private** onMouseLeave(){  
 **this**.hightlight(**'null'**);  
 }  
 **private** hightlight(color:**string**) {  
 **this**.el.**nativeElement**.**style**.**backgroundColor** = color;  
 }  
}

### Configure the directive with binding

We should set the color externally with a binding like this:

<p [myHighlight]="color">Highlight me!</p>

We'll extend our directive class with a bindable input highlightColor property and use it when we highlight text.

**import** {Directive, ElementRef, HostListener, Input} **from "@angular/core"**;  
@Directive({  
 **selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
 **private defaultColor**:**string** = **'red'**;  
 **constructor**(**private** el: ElementRef){}

*//@Input(alias) alias the highlightColor property with myHighlight*  
 @Input(**'myHighlight'**) **highlightColor**:**string**;  
 @HostListener(**'onmouseenter'**)  
 **private** onMouseEnter(){  
 **this**.hightlight(**this**.**highlightColor** || **this**.**defaultColor**);  
 }  
 @HostListener(**'onmouseleave'**)  
 **private** onMouseLeave(){  
 **this**.hightlight(**'null'**);  
 }  
 **private** hightlight(color:**string**) {  
 **this**.el.**nativeElement**.**style**.**backgroundColor** = color;  
 }  
}

Update the template:

<h1>My First Attribute Directive</h1>

<h4>Pick a highlight color</h4>

<div>

<input type="radio" name="colors" (click)="color='lightgreen'">Green

<input type="radio" name="colors" (click)="color='yellow'">Yellow

<input type="radio" name="colors" (click)="color='cyan'">Cyan

</div>

<p [myHighlight]="color">Highlight me!</p>

### Bind to a second property

Let's allow the template developer to set the default color, the color that prevails until the user picks a highlight color. We'll add a second input property to HighlightDirective called defaultColor:

@Input() **set** defaultColor(colorName: **string**){  
 **this**.**defaultColor** = colorName || **this**.**defaultColor**;  
}

In html-template:

<p [myHighlight]="color" [defaultColor]="'violet'">

Highlight me too!

</p>

# Words

**cumbersome** 英 **['kʌmbəsəm]**  美 **['kʌmbərsəm]**  **adj.笨重的；不方便的**

**cumber** 英 ['kʌmbə]   美 ['kʌmbə]  v.**拖累；妨碍** n.**妨碍；累赘**

**onerous** 英 ['əʊnərəs]  美 ['ɑːnərəs] adj.**繁重的**